MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I.Memo No. 977 July, 1988

A Standard Architecture for Controlling Robots

Sundar Narasimhan
David M. Siegel
John M. Hollerbach

Abstract: This paper describes a fully implemented computational architecture that controls the Utah-MIT dextrous hand and other complex robots. Robots like the Utah-MIT hand are characterized by large numbers of actuators and sensors, and require high servo rates. Consequently, powerful and flexible computer architectures are needed to control them. The architecture described in this paper derives its power from the highly efficient real-time environment provided for its control processors, coupled with a development host that enables flexible program development. By mapping the memory of a dedicated group of processors into the address space of a host computer, efficient sharing of system resources between them is possible. The software is characterized by a few simple design concepts but provides the facilities out of which more powerful utilities like a multi-processor pseudo-terminal emulator, a transparent and fast file server, and a flexible symbolic debugger could be constructed.

Acknowledgements: This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Office of Naval Research University Research Initiative Program under Office of Naval Research contract N00014-86-K-0685 and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

1 Introduction

This paper describes the *Condor* system, a real-time control environment designed for robotics research applications. The *Condor* system was derived from the *Muse* (Narasimhan et al. [1986]), an earlier system also developed at MIT.

Sophisticated control strategies, advanced robot designs with more degrees of freedom, and high performance actuation and sensor systems all add to the computational needs of a robot controller. Without advanced computational architectures, it is impossible to develop and test more than the simplest of control strategies.

In our laboratory, the Utah-MIT Dextrous Hand pushed these limits in all directions. The hand has 16 joints, each powered by two antagonistic tendons connected to electro-pneumatic actuators. There are 32 tendon tension sensors and 16 joint position encoders. Typically, a servo rate on the order of 400 hertz is required. This necessitates reading 19,200 sensor values per second and writing 12,800 actuator values per second. An examination of robot controllers developed at our laboratory and elsewhere revealed that conventional uni-processor based controllers would not be suitable for this device.

Any research oriented real-time controller must be both flexible and efficient. A high performance controller that lacks sufficient development tools will be hard to use, and is not suitable for a research environment. A system with a good development environment, but lacking in computational performance, will not handle the needs of robots being built in research laboratories today. Computational architectures found in industry often provide performance at the expense of flexibility, and often lack an adequate software development environment. University efforts have often resulted in architectures that do not provide adequate real-time response. The *Condor* attempts to meet both these needs by combining a real-time processing engine with a conventional development host.

Computers alone cannot make up for inefficient algorithms. Even the fastest processors may not be able to execute poorly designed control algorithms at the required speeds. Thus, the control of a robot requires both adequate processing power and efficiently formulated algorithms. For example, Hollerbach et al. [1986] formulate a finger force computation scheme for the Utah-MIT Hand that greatly reduces the computations required by the standard Grip-Jacobian approach.

Initially, the *Condor* was designed to control the Utah-MIT Hand (Jacobsen et al. [1984]). However, the architecture is sufficiently general to control other robots. At MIT, this system is being used to control the MIT Serial Link Direct Drive Arm and the Whole Arm Manipulator. It is also being used to run biological motor control experiments, since the *Condor* is a general real-time system useful outside of robotics. Additionally, the *Condor* is being used by other research groups that have obtained the Utah-MIT Hand.

This paper provides an overview of our architecture and how it is being used to control the Utah-MIT Hand. Since this is a fully implemented project that has been in operation on several robots for over a year, we feel our experiences will be of interest to other researchers who face similarly complex control needs.

1.1 Special purpose vs. general purpose hardware

The spectrum of choices available for robot controller architectures ranges from using off the shelf commercially available systems to building special purpose VLSI chips tailored to specific algorithms (Leung et al. [1987]). The *Condor* is designed to control *research* robots. Control algorithms for such robots are often as intense a subject of research as the design of the robots themselves. The programmability required of such control systems coupled with the evolving nature of their control algorithms dictated the use of general purpose micro-processors for the *Condor* system.

There were many reasons why we decided not to build special purpose hardware for the *Condor* . Firstly, we wanted a standard computer platform on which many research robots could be built, all sharing the same base of controller software. Using a standard bus ensures the availability of peripherals for interfacing analog to digital and digital to analog converters, resolvers, and shaft encoders. Secondly, the amount of effort invested in building and maintaining custom hardware is often disproportionate when compared with the increased performance they provide. Most university robotics laboratories are not staffed by hardware designers, but rather by researchers whose primary focus is robotics. Hardware development is therefore best left to organizations that specialize at it.

We do not wish to imply that there is no place for special purpose hardware in robotics. Time and space constraints and very tight performance requirements may necessitate custom VLSI implementations in certain cases. There is a trend nowadays toward using ASIC based hybrid designs and special purpose DSP hardware to speed up critical computations involved in robotics. The *Condor* is sufficiently open-ended to facilitate the addition of suitable hardware speedups as they become available. We feel such additions will be useful if they are determined to be essential once the control algorithms have matured sufficiently. Such chips are inadequate, however, to form the basis for an entire development environment at an early stage.

1.2 Multi-micro processor based systems

Given the above remarks, there are a number of different "standard" choices on which one could base a robot control development environment (see Leung et al. [1988] for a survey of possible alternative designs). We feel the power delivered by present day microprocessors is quite adequate for controlling most robots.

- *Coarse parallelism.* Robot control is characterized by a relatively small number of computations that run in parallel, such as joint servo loops. Hence highly parallel architectures are unnecessary.

- *Efficient algorithms.* The computational requirements for implementing robot kinematics, dynamics, and control algorithms are just not that severe [Hollerbach, 1988]. Current microprocessors are adequate for this task, and there is no need for special-purpose architectures.

Other researchers who have echoed a similar view include Bisiani and Reddy [1982] and Brooks et al. [1984].

Essentially, the *Condor* is comprised of a development host connected to a real-time controller. The former offers all the facilities for program development while the latter is a tightly coupled multi-microprocessor based environment. The Motorola 68020 processor is used in both components of the system to insure a high degree of software and data compatibility between them. When computational requirements increase, additional power can be obtained simply by adding more processor boards to the system and repartitioning the control algorithms.

Other researchers have also recognized the advantages that such multi-micro processor based controllers can provide in terms of price, performance, and programming ease (Chen et al. [1986], Paul and Zhang [1986], Kazanzides et al. [1987]). A recent survey article by Gauthier et al. [1987] contains information on other robot control architectures based on microprocessors. Our system differs from many of them in two important ways:

1. We attempt to avoid duplicating software and hardware components that can be found on conventional, non real-time, systems.

2. We provide an extremely efficient computation environment.

Only the minimal set of features necessary to provide a reasonable and convenient environment were included. More complex facilities are always available from the host computer. This insures highly efficient operation of the *Condor* 's controller component.

For example, Kim, et al. [1987] describe a Multibus based real-time multiprocessor system for controlling the Puma/RAL hand system. Their development environment runs on one of the controller processors and provides access to disk and other system resources. We chose not to use this approach for several reasons. Most importantly, we feel that conventional computer systems offer *superior* file serving and user interface capabilities, and we did not want to duplicate such facilities in our real-time system. Secondly, bus bandwidth on the real-time system is best left for real-time uses; adding the development host's traffic on the same bus only exacerbates the contention for this resource.

2 The Hardware

Our hardware system is pictured in Figure 1. In this design, a Sun-3 system forms the main development machine, while the real-time controller utilizes Ironics Corporation Motorola 68020 based single board computers each equipped with a Motorola 68881 floating point unit. The control processors are linked using a VME bus and are coupled to the development host through a VME bus to VME bus adaptor. The real time bus is separate from the bus on which the development host resides.

2.1 The *Condor* system hardware

There are a number of desirable features in this hardware design which we highlight in the following discussion:

1. *Industry standard interconnect*: The high speed VME bus has become an industry standard for control applications. The advantage of using such a standard bus is

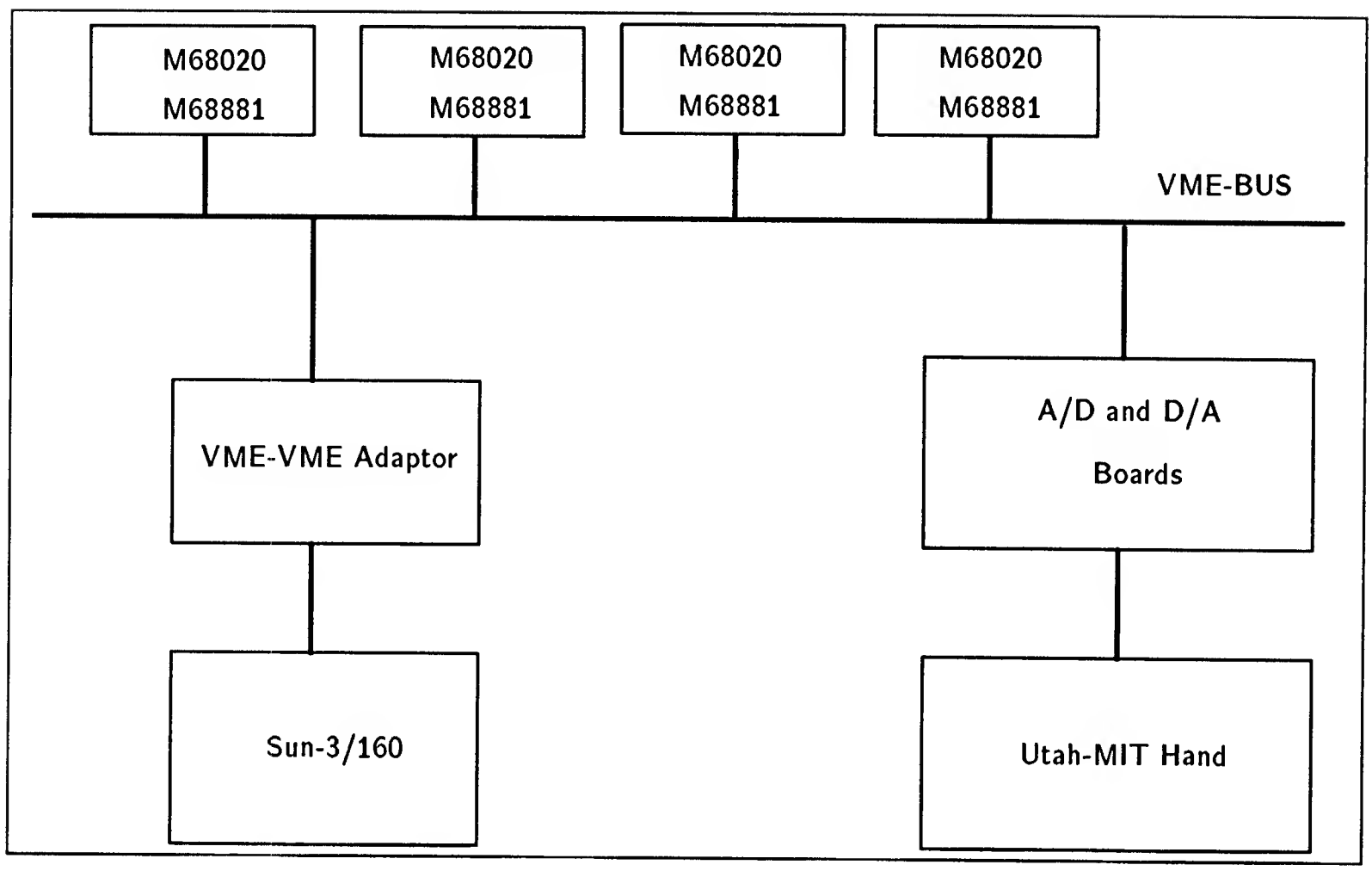

Figure 1: *Condor* hardware block diagram

the availability of CPU and peripheral boards from a number of vendors. This is an important capability needed for robots whose sensors and actuators often require a diverse array of interfacing capabilities.

2. *Bus to bus adaptor*: The bus-to-bus adaptor permits transparent access from the development host into the dual-ported shared memory on the control microprocessors. Downloading of control programs, for example, becomes equivalent to an array copy operation. The adaptor also enables programs that collect data to run on the development host with access to disk files, without interfering with the tasks involved in real time control.

   In an earlier version of our hardware, a DMA connection was used to communicate between the host processor and the control microcomputers. The DMA connection was found to be of limited use due to the high overhead of each transfer. Though a DMA connection has the desirable property of isolating the connected buses when transfers are not occurring, the performance overhead that this incurs in a real-time control environment is unacceptable. In particular, a DMA connect precludes shared memory pointer access from one system into the other. Though message passing over the DMA link can be used to move data rapidly, a shared memory access incures much less overhead, and is desirable in many controller applications.

We decided to use a bus to bus adaptor to connect the development and real time hosts because communication hardware like RS-232 serial lines, or parallel interfaces would have been too slow and devices like ethernet controllers would have required too much overhead to set up and complicated software to be utilized effectively.

3. *Real time host and development host compatibility*: Both the Sun-3 and the Ironics computers are based on the same Motorola 68020 microprocessor. This avoids the need for cross-compilers and special-purpose linkers and loaders. This also enables debugging of most programs on the development host, which reduces time spent in the download, run, debug cycle. The separation of the real time and development environments has also allowed us to optimize the former for extreme efficiency and the latter for ease of writing and debugging control programs.

4. *Speed of the single board computers*: The individual microprocessors are fast processors, and coupled with a floating point engine they provide a much faster environment for scientific and control computing. We decided against some of the more recent RISC CPU's and other products under development primarily because of software considerations. The *Condor* needed to be based on stable optimizing compilers and needed to have an environment which could be programmed in a conventional programming language.

5. *Development host:* The Sun-3 development host provides a very good environment in terms of bit-mapped graphics, an industry standard window system, and a transparent compiler (in the *Muse* a cross compiler had to be used) that generates compact code. Code that runs on the controller processors can typically be run on the Sun-3 development host without recompilation. Simply relinking the procedures with a Sun version of the run time library is all that is required.

Based on our experience with two versions of our design, we outline below hardware capabilities that we view as important for any architecture proposed for a multi-micro processor based robot controller.

(a) *Support for a mailbox interrupt:* This capability is used for message passing based interprocessor communication. It provides the mechanism for one processor to interrupt another, indicating that a message is being sent between them.

(b) *Dual ported memory:* used for interprocessor communication, shared data structures, and host to microprocessor utilities. Allows highly efficient access of shared data, unlike a DMA, parallel, serial or ethernet connection which necessitates transfer of data.

(c) *Interrupt generation unit:* used for interprocessor communication between microprocessor and host computer. Provides a mechanism for the microprocessors to interrupt the Sun, which lacks the mailbox interrupt mechanism.

(d) *Floating point unit:* eases development of control code.

(e) *Timer interrupt generator:* used for control loop scheduling. Generates a regular interrupt source for use in executing control code.

(f) *Test and set instruction:* used for interprocessor communication. Allows multiple processor to access shared data structures in a controlled fashion.

3 System Software

An important part of any computational architecture is the software that is available to run on the hardware. The previous comments regarding flexibility and efficiency of robot control hardware apply equally as well to the software components. In this section we will provide an overview of what we feel are the innovative aspects of this part of the *Condor* system.

The design goals of the software system were to:

(a) provide a flexible environment in which control programs can be written, debugged, and run,

(b) provide efficient and low overhead means of doing often repeated tasks,

(c) provide easy to use programmer libraries for data transfer hardware, and for real-time interaction with robotic devices, and

(d) provide a user interface with bit-mapped graphics.

To achieve these goals, the *Condor* system is structured around a few relatively simple organizing principles. In a typical program development scenario, the user is expected to write and compile a program on the development Sun-3 host. Since the Sun runs the Unix operating system[1], the programmer has access to all the standard Unix software development tools. Once the program has been compiled, it is linked with the real-time library, and then downloaded onto the slave microprocessors for execution. The run-time environment provides the user with access to a number of program libraries for performing common tasks in a portable manner. For example, libraries are provided for control loop scheduling, file serving, and plotting data, to name just a few. Besides these, libraries are also provided for mathematical and control computations.

Thus, the *Condor* environment is *two* different environments. First, there is an environment on the Sun known as the *development* environment, and then there is the *run-time* environment for both the Sun and the slave microprocessors. Much effort has been put into making the Sun and the slave microprocessor run-time environments as compatible as possible. In fact, most programs that run on the real-time controllers will run on the Sun simply by relinking them with the appropriate library.

The hardware architecture is tightly-coupled, and is a true MIMD machine. As such, it requires a program to be partitioned into segments that can run on multiple processors. Our approach to managing this multiplicity problem in terms of software engineering has

[1]Most of our system software runs on Sun 0S versions 3.4, which are closely compatible with and are based on Berkeley 4.2 BSD Unix.

been to largely ignore it. The number of processors in a *Condor* system rarely exceeds five or six and we do not expect this architecture to be applied to problems requiring more than a dozen processors. In essence, the course-grain parallelism applied to control programs is managed by the programmer directly.

Typically, control programs can be partitioned fairly easily. For example, in the digital controller for the Utah-MIT Hand, one processor is dedicated to controlling two fingers. Should the controller's complexity increase to the point where one processor is required to control each finger, a manual repartitioning of the computations would be required. By careful use of message passing, however, it is possible to design code that can easily be reconfigured for different partitioning schemes. Managing the different programs that need to run on the different processors can be done with a few simple rules of thumb and with existing software tools like **make**.[2]

Thus, the *Condor* system as a whole consists of

(a) a tightly coupled MIMD processing engine, linked to a Sun with a VME-VME memory mapped adapter,

(b) Sun window based interface to the microprocessor controllers,

(c) Sun window based plotting and other analysis tools,

(d) a Sun version of the run-time library, and

(e) a microprocessor version of the run-time library.

The following sections provide an overview of some of these components in more detail. More detailed documentation on these functions can be found in the working paper titled *The Condor Programmer's Manual* (Narasimhan et al. [1987]).

3.1 Devices

Interacting with robots usually requires interfacing a controller to various input and output devices. Most robots have idiosyncratic front-end controllers, and the array of sensors connected to them is diverse. A computational architecture must support both easy hardware interconnections and easy software interface to these external devices. Since the *Condor* is based on the VME bus, hardware interfacing is straightforward. Software integration utilizes a *device switch* structure modeled after the system used by the Unix kernel.

The design of the device system was motivated partly by experience. In the *Muse* , there was no systematic way of accessing devices. What existed was an ad-hoc interface between the Unix-style **read** and **write** calls and various device specific routines for the controller interface hardware. To use a parallel port board a user had to know its particular initialization routine, and often needed to know such details as the device's control register address. In the *Condor* we replaced this with a clean design from the lowest level.

[2] **make** is a Unix utility for managing programs built from a number of different modules.

The *Condor* real-time environment was designed to provide an extensible way of writing device drivers. The low level details of a device, including its register formats and interrupt mechanisms, are abstracted from user code and hidden within the device driver. The trick was to keep the device drivers highly efficient, while providing the necessary level of abstraction to free user code of low level details. The added expense incurred by a standard operating system's device driver mechanism would not be acceptable in our real-time controller environment.

The *Condor* system's device mechanism is designed to be extremely fast and portable. The mechanism is static. No support for dynamic loading of device drivers is provided, which necessitates recompiling the system libraries each time a new hardware device is installed. The overhead associated with recompilation is significantly lower than the overhead and complexity associated with any dynamic loading scheme.

These mechanisms provide functions that:

(a) automatically initialize devices,

(b) handle interrupting devices,

(c) handle shared interrupt vectors,

(d) emulate system calls like `open`, `read` and `write`, and

(e) handle devices that may require more than the standard read and write style accesses.

Though the mechanism is modeled after the style found in early versions of Unix, there are a few significant differences. Each device is essentially modeled as an abstract data type, on which a few standard operations can be performed. When such a device is opened, an integer object called a `file descriptor` is returned whose semantics are close to that of the conventional Unix file descriptor. This object can be used as an argument to other operations that are performed on the device.

The following code fragment shows how a parallel port is opened, and used:

```
int
parallel_port_startup(board_number)
int board_number;
{
    int fd;
    if((fd = open(``:mpp'', board_number, 2)) < 0){
        printf(``Couldn't open device?\r\n'');
        exit(0);
    }

    /* Reset the board */
    mpp_reset(fd);

    /* Configure the board to be in raw 16-bit mode */
```

```
    mpp_config_16bit_raw(fd);

    /* return the fd, so that the user can use it later */
    return(fd);
}

parallel_port_read(fd, buffer, count)
int fd;                 /* fd associated with the board */
short buffer[];         /* Buffer passed in by user */
int count;
{
    register int i;
    short value;

    for(i=0;i<count;i++) {
        /* Perform the read 16 bits at a time */
        buffer[i] = mpp_read_16bits(fd);
    }
    return(count);
}

parallel_port_exit(fd)
int fd;
{
    return(mpp_close(fd));
}
```

As the above example illustrates, each device has an **open** routine and a **close** routine. There is one system-specific configuration file that tells the run-time system the types of devices that may exist. The *Condor* run-time system will determine upon startup, which of those possible devices are actually present and initialize them using their device specific **init** routines. This is analogous to the **probe** routine, used by Unix systems.

Device independent operations like **open**, **read**, **write**, and **close** are mapped to device specific routines using the supplied file descriptor and the device switch mapping table. In addition, a device-specific buffer is allocated for each opened device, and contains data that is used internally by the device system. The lower level interrupt routines operate on these device-specific buffers.

There are a number of operations performed on devices that do not easily fall into the Unix read and write paradigm. The standard Unix way of handling such unusual devices is to overload the **ioctl** system call. In the *Condor* system we chose to use the following conventions, and in practice, this has proved effective.

1. Device specific routines are uniquely named by prefixing the routine with the name of the device (for example, a routine for configuring the **mpp** device will be called **mpp_configure**).

2. Routines that are peculiar to a device will take the file descriptor as the first argument and map it to a device specific structure. Once the mapping has been made the driver can perform any necessary operations. This essentially provides entry points into the device driver bypassing the standard device structure, and eases the task of writing modular device drivers.

The device driver interface also provides the low-level glue for the interrupt mechanisms, the file server interface, and the buffered input and output libraries (commonly known as `stdio` under Unix).

3.1.1 Interrupts

Another capability that control system architectures require is servicing interrupts in real-time. These interrupts usually correspond to events that require attention, or periodic interrupts from timers.

The VME bus provides support for eight levels of prioritized vectored interrupts, and the Motorola 68020 processor has the capability to support 256 different interrupt vectors. Interrupts on the VME Bus are vectored, in contrast to the Multibus which typically supports only non-vectored interrupts. The Multibus-II supports a different notion of interrupts which essentially increases the number of different levels of interrupt available on the bus, and is in some ways more desirable than the scheme supported by the VME bus. However, VME bus compatibility with the Sun-3 hardware was considered to be more important.

In real-time control, interrupts can come from a variety of sources: interval timers, analog to digital and digital to analog converters, parallel and serial devices, etc. A uniform way in which all interrupts are handled is indispensable in a development system. In the *Muse* , interrupts were handled using a number of different stubs all written in assembler. In the *Condor* , all interrupt vectors map to the same higher level routine. The system has a software data structure that maps vector numbers to interrupt servicing routines. This data structure is used to map incoming interrupts to their appropriate servicing routines. This scheme has resulted in a single assembler routine that services all interrupts.

There are a few complications that the system must handle. A raw interrupt event must be mapped to a file descriptor corresponding to a device. Since the *Condor* is not multi-tasking, no distinction exists between system space and user space. When a serial port interrupts the system, the device generic interrupt handler must determine which serial driver's input buffer should receive the character.

The problem is further complicated by shared interrupt vectors, where multiple devices can interrupt the system with the same interrupt vector. The *Condor* system solves this complication by maintaining a list of all devices that receive interrupts on a particular vector. When more than one device uses the same interrupt vector, the *Condor* system maps this interrupt vector to a generalized *device-level interrupt* vector. This routine searches a list of interested devices and invokes the interrupt routine of each of those devices one by one, polling the possible choices.

3.2 Scheduling

Real time control involves scheduling events at specified servo rates. For example, most robot control involves loops that read data from analog to digital converters, perform some control law computation, and then write out computed values to the actuators using digital to analog converters. These low level control loops need to be scheduled so as to run at specified sampling rates. The *Condor* provides two such mechanisms by which servo loops can be scheduled on the real time microprocessors. Both mechanisms rely on scheduling at the C procedure level; the user has the ability to specify that a particular C procedure has to be executed at a specified rate. Other systems that provide scheduling at a higher level (task or process level) suffer from higher task-switching overhead and hence lower real-time performance.

3.2.1 Simple Servos

The simple mechanism for scheduling servo loops on the *Condor* provides support only for two running tasks on a single *Condor* micro processor. One of these tasks is the control loop that is scheduled to run at a specified rate, while the other is a background task that runs whenever the control loop is not running. This background task is usually dedicated to running a command loop that interacts with the user, and can perform various functions like enabling or disabling the real time task. The background task can also reschedule the real time loop to run at different rates. This simple mechanism has very little overhead.

3.2.2 The MOS

For users that need to schedule more than one real-time loop to run on a single *Condor* microprocessor, a more complex interface called the MOS is provided.

This second set of library routines allows a processor to run various control loops at different rates, in a highly efficient manner. Since a typical hand control program will have several servo loops running at rates in excess of 400 Hertz, it is important for each scheduler invocation to be fast. To achieve this, scheduling flexibility has been limited to minimize the execution overhead that it requires. In fact, it is a gross overstatement to call this an operating system. It is, in fact, just an efficient utility for programming a system timer and for starting procedures based on precomputed rate information.

To minimize execution overhead, the MOS is table driven. An *event table* is automatically generated by the system when the `mos_start` command is issued. This table lists the elapsed time between invocations of the scheduled servo loops. For example, the event table for two servo loops, one running every ten seconds and the other running every five seconds, has two entries. The first entry indicates that both loops are to start, and five seconds elapse until the next event. The second entry indicates that the five second loop should start, and another five seconds are to elapse before the next event. After this, the cycle repeats, and the first entry of the event table is reused.

With the system outlined so far, it is possible for more than one loop to be runnable at the same time. The system must have an orderly method for selecting the actual loop that will be run from the set of runnable loops. A *process table* is maintained for this purpose.

All the tasks in the system are arranged, in order of decreasing servo rate, in this table. When the event table indicates a loop is ready to run, it is marked runnable in the process table. The system then searches down the process table, and starts the fastest rate loop that is marked runnable.

The time to the next event stored in the event table is loaded into a timer on the processor. When the time has elapsed, the running task is interrupted, and the scheduler is reinvoked. The next tasks in the event table that are scheduled to start are marked runnable in the process table. If a loop with a speed slower than the interrupted loop is made runnable, the interrupted loop will be resumed. If a higher speed servo loop is made runnable the slower loop that was interrupted will be temporarily suspended, until higher speed loops that are runnable complete.

An implication of assigning a priority to a process based on its rate is that a loop can only be interrupted by a higher speed loop, and hence, no coprocessing can take place. This is not considered to be a problem. The rate specified for a servo loop is a request that the loop be run that number of times a second. The exact time that a loop is invoked is not important, as long as it is runs within its specified time slice. In other words, a loop scheduled to run every second is only a guarantee that the loop will run *sometime* within a second. A finer precision in selecting the time at which a procedure will run is not needed within our control programming scenario.

Eliminating coprocessing results in a convenient simplification to the system; only one stack need be maintained for all the servo processes running on a processor. Stack pointers are not changed when a new process is invoked, or a suspended process is resumed.

When a loop terminates, the scheduler is also invoked. The terminating loop is marked idle in the process table, and a new loop is selected to run. If no servo loops in the process table are runnable, the background job is activated.

We have found that the mechanisms described above satisfy most of our requirements when it comes to scheduling real time servo loops on the *Condor* .

3.3 Message Passing

While device drivers and other utilities provide support for bootstrapping and running a program on a single processor, the *Condor* message passing system addresses the issue of multiple processors and communication between them.

The message passing system provides a simple and low-overhead manner in which communication of data can occur between multiple processors, and between processes on the Sun. Since robot control is always compute bound, a system for communication between such tasks has to be extremely time-efficient. The primary design goal of the *Condor* message passing system was therefore efficiency.

Interprocess communication for robotics, as mentioned by Gauthier et al. [1987], has been tackled in a variety of ways. Architectures based on RS-232 serial lines and parallel ports rely on framed protocols while those based on ethernet devices rely on packet switched protocols for their communication needs. In tightly coupled systems, shared memory based schemes are more popular than message passing schemes.

The present system is to a large extent a redesign of the system described in Narasimhan

et al. [1986], which was more flexible but less efficient. The newer implementation essentially uses shared memory to implement a highly efficient message passing based scheme.

3.3.1 Messages

Since the Ironics processors and the Sun host computer are all bus masters on a common VME bus, each machine has access to each other's dual-ported memory. Interprocessor communication occurs over the bus and uses shared memory. This allows any processor to directly access data in another processor's memory. The most basic form of interprocessor communication possible would be direct memory reads and writes. Unfortunately, while this unrestricted access is highly efficient, it is hard to control.

To overcome the problems of unrestricted memory access, a mailbox-based message passing system is supported. Mailbox interrupts can be thought of as a software extension to the processor's hardware level interrupts. Another way of thinking about them conceptually is to regard mailbox numbers as port numbers that map to specific remote procedure calls.

A mailbox interrupt has a vector number and a handler routine. When a particular mailbox vector arrives, its appropriate handler is invoked. The handler is passed the processor number that initiated the mailbox interrupt and a one integer data value. This integer data value is the message's data[3].

The message passing system in the *Muse* was substantially more complex. Messages could be of arbitrary size, and they were addressed to virtual devices that corresponded to the mailbox handler routines in the present version. These handler routines were assigned to processors by a preprocessor that took as input an assignment file, that configured the routines available on each processor. The preprocessor then generated a routing table that had to be linked in with each program. The routing table mapped a virtual device number to a processor that could handle the function. The *Condor* redesign was done because the complexity and the overhead of the earlier system prevented most control programs from using message passing.

An important capability that the earlier implementation lacked was the ability to reply to messages. A program could not determine if a particular message succeeded or failed. Implementation of messages with replies could be done in the *Muse* only through an ad-hoc mechanism that was very inflexible. The *Condor* provides support for messages with or without replies. The implementation uses a reverse send from the recipient processor to the sending processor to acknowledge the receipt of a message. The implementation has been written carefully to be reentrant so that nested sends will work correctly.

The following example illustrates the basic facilities provided by the system. In the example, a message will be sent to a handler to read the value of a memory location. The location to be read is passed to the handler as the data portion of the message. The reply from the handler is the contents of that memory location. The code fragment for the handler would be:

```
simple_reference(proc, data)
int proc;
```

[3]Integers are currently any 32 bit data quantity.

```
int data;
{
    return(*(unsigned int *)data);
}
```

The handler is associated with a vector number using `mbox_vector_set`:

```
mbox_set_vector(12, simple_reference, ''A test handler'');
```

Now, any message sent to this processor for vector 12 will be handled by the `simple_decoder` handler.

Another processor can invoke the decoder by using the `mbox_ send` routine. If the `simple_reference` routine is available on processor 0 one can execute the following piece of code on any of the processors (including 0 itself) to invoke the service.

```
value = mbox_send_with_reply(0, 12, address);
```

This will cause the handler that corresponds to the number 12 to be invoked on processor 0 with the second argument being `address`. The call will not return until the other processor has responded with the value found at the given address. This call can be used to provide synchronization. For services that do not require synchronization, and hence do not return a value, the `mbox_send` call can be used.

In summary, the following are the key features of the message passing system:

(a) Since message sending happens asynchronously, the execution of a handler resembles an interrupt. Real time computation is based on a timer interrupt whose priority is higher than that of the mailbox interrupt. Consequently, mailbox handlers have to be written so as to be interruptible.

(b) The base system is extensible in the sense that more complicated protocols can be built on top of it. For example, the underlying system does not support queueing of messages, although one can easily build one for mailboxes that require this.

(c) Since the message system is based on shared memory, sending long messages is usually handled by sending a pointer to the beginning of a long piece of data.

(d) Where efficiency is important, the message handling system can be used to set up pointers from one processor into another's memory. The processors can read and write this shared memory, without the minimal overhead of message passing.

Message sending and the invoking of message handlers is implemented using a *mailbox interrupt* which is a hardware interrupt that is invoked by writing into a particular memory location in a processor's memory. This hardware support is critical for the implementation's efficiency. To protect the integrity of certain critical data structures the *test-and-set* instruction is used. It is important that this instruction be supported truly indivisible by the hardware across the bus.

Table 1: Performance of the Message Passing System. (R) refers to messages with replies.

Type of Operation	Msecs/Message
Ironics to Sun	34 ± 5
Ironics to Sun (R)	38 ± 10
Sun to Ironics	3.9 ± 0
Sun to Ironics (R)	4.0 ± 0
Ironics to Ironics	0.2 ± 0
Ironics to Ironics (R)	0.25 ± 0

3.4 Support for Message Passing on the Sun

The Sun development host supports the same primitives for message passing available on the control microprocessors. Any number of processes on the Sun may communicate with each of the slave processors using the message passing protocol.

A message that arrives on the Sun must be mapped to a particular Unix process that can handle it. While each microprocessor is thought of as a single message-receiving processor, the Sun supports the notion of *virtual processors*. Each Sun process that receives interrupts is assigned a unique hardware interrupt vector. Each process on the Sun which participates in message passing must register itself with the Sun kernel, indicating which hardware interrupt vector corresponds to its messages. When a slave processor interrupts the Sun it does so using this vector. The Unix kernel traps on this interrupt vector and signals all processes that have expressed an interest in receiving the interrupt.

From the Sun, the *Condor* system maps the entire VME bus 24 bit address, 32 bit data (VME24D32) space into the user address space of the control process. Memory references to any of the control processor's memory, or to the additional one megabyte memory board allocated in the VME backplane for Sun use, become simple array references. The `PROC_RAM` macro returns a pointer to the beginning of memory for the particular processor. For example, to write a value to location 100 in processor 3's memory one would use the following code:

```
int *processor3_ram = (int *)(PROC_RAM(3));
processor3_ram[100] = value;
```

The `PROC_RAM` macro is also used for programs running on controller processors to access memory of other Ironics processors. The code above would work, in fact, on any processor in the system.

Table 1 summarizes the performance of the message passing system as benchmarked by a variety of routines. As can be seen from this table, the performance of the message passing system is extremely fast between two control processors. The slower speed for messages sent from the control processors to the Sun host is caused by overhead present in the Unix timesharing operating system.

It should be noted that the message rates are not as high as servo rates. Consequently, messages are used only as signals to start and stop processes or control the flow of computation and are not used as timing pulses.

3.5 Message Passing and its Implication for Control

Using the facilities provided by the message passing system it is possible to treat a hierarchical controller as an object-oriented system that responds to control messages. For example, a low-level joint PID controller could be described as an object that responds to two different kinds of messages: messages that alter internal parameters like gains and position set points, and messages that control the execution of the servo loops. Using this scheme, any processor, including the Sun host, can control the execution of any servo loop in the system.

3.6 Higher Level Protocols

The *Condor* system requires the concurrent operation of several control microprocessors to perform its tasks. Programming such a complex MIMD machine would certainly be a nightmare were it not for the numerous services that were built on top of the base system using the message passing facilities. These services provide flexible interaction with a number of slave microprocessors at a time, file server capabilities on the development host, and symbolic debugging.

3.6.1 Debugging with ptrace()

One of the most important utilities in the system is the symbolic debugger. To implement a debugger in a flexible manner it was decided to emulate the Unix `ptrace` system call. `Ptrace` is used by Unix debuggers to examine registers, set breakpoints, and control program execution of a slave process. Our emulated `ptrace` is linked into all control programs as part of the standard library.

The emulated `ptrace` communicates with the Sun-3 development host using the message passing system. For example, if the Sun-3 host wishes to examine a control program's registers, it would send a message down to the processor's `ptrace` handler. The `ptrace` handler would reply with the desired information. The debugging program runs on the development host. Only the low level `ptrace` routine runs on the microprocessors. It is conceptually useful, therefore, to think of debugging messages as being passed between two processes; a *slave* `ptrace` process that runs on the control microprocessors and the debugger that runs on the development host.

By emulating the low level `ptrace` call, almost all Unix based debuggers can be adapted, with little modification, to debug programs on the controller microprocessors. Currently the *Condor* uses the Gnu Debugger (GDB). This debugger provides fully symbolic, C source code level debugging tools. It has been used to debug high level application programs, and to find and fix low level system bugs.

The debugger can be used in three basic modes. The first allows the user to start executing a program directly under the debugger's control. This is done by beginning the execution of a downloaded program at a different location that transfers control to the debugger instead of the usual start point. The program can then be single stepped or continued by the debugger on the development host.

The second mode may be used when a bug is actively being tracked down, and when setting initial break points might be necessesary. This mode allows the debugger to be attached to a processor after execution has begun on the control microprocessor. For example, if a program were in an infinite loop, the debugger could be attached to the running program to determine what went wrong.

The third mode is used by default when a program receives a fatal exception that is considered to be a debuggable error. For example, addressing errors, bus errors, and various floating point exceptions would cause the errant program to be stopped. Control would be transferred to the `ptrace` slave process which will then wait for a debugger to be invoked on the development host.

Once the debugger has attached to a process on the slave control processor, all the capabilities of the debugger can be used to debug the program running on the remote host, just as if it were running on the development host. This scheme permits the best debugger available on the development machine to be used on programs running on the control microprocessors with very little effort.

3.6.2 File Serving

Control programs often need access to data stored in files. Experimental data collected during experiments also need to be stored in disk files to facilitate further analyses. To allow these types of access, a file server protocol, built using the low level message passing system is provided. From a control program point of view, the standard Unix file operations are available, both buffered and unbuffered. Typically, each file operation results in a message being sent to the Sun, where a server process performs the equivalent Unix file operation. The result is then sent back to the microprocessor as a reply.

To make the file server efficient, shared memory is extensively used to avoid copying data. When a microprocessor performs a read or a write, the file server process running on the Sun reads or writes the data directly to or from the microprocessor's memory. No intermediate data copy is required. To perform a read, for example, the microprocessor would send a message to the Sun giving the address to read the data into, and the number of bytes to read. The data is directly transferred into the processor's buffer.

The file server is designed to operate in a stateless manner. All necessary data is stored on the microprocessor. The Sun server does cache some information, but if necessary, it can request the information from the microprocessor. Thus, even if the Sun file server process were to be terminated and restarted, the microprocessor can continue its operation without interruption.

3.6.3 Virtual Terminals

Many microprocessors are used in a *Condor* system. The Sun-3 host computer provides a window-system based interface to access these computers. One window for terminal input and output to each computer is provided. A virtual terminal protocol, built using the message passing system, routes data between the Sun and the microprocessors. This avoids the bank of terminals that would otherwise be connected to the real-time controller.

The virtual terminal protocol works in both directions. The Sun can send a message to a microprocessor that contains an input character. The character is added to the terminal input queue by the virtual terminal message handler. Each character is sent in one message, since transfers in this direction are limited by the rate at which a person can type.

Terminal output from a microprocessor to a Sun virtual terminal window is sent in blocks, and not one character at a time. The output message contains a pointer into the terminal driver's memory, and the number of characters to output. Such a virtual terminal protocol allows a programmer to have independent windows assigned to each of his processors through which he can interact with each of his programs running on those processors.

3.7 The Condor User Interface

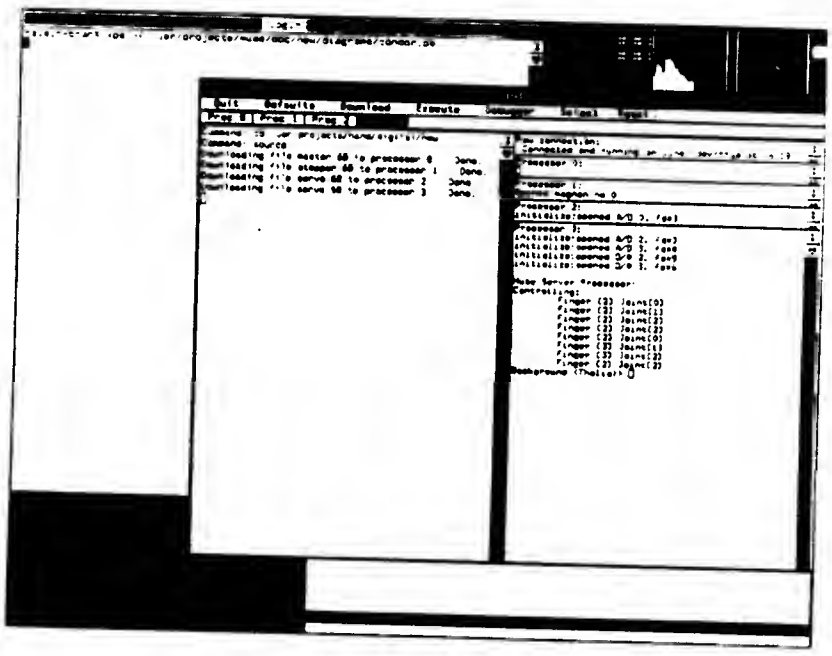

Figure 2: The Condor running under the X Window System

The file server, debugger and virtual terminals combined together form the *Condor* user interface. This program is an X-window system based application that programmers utilize to interact with the slave microprocessors. The user interface provides one virtual terminal for each slave processor, it runs a file server, and it can start debuggers and other programs. Figure 2 shows the screen of a typical *Condor* user interface session. This user interface program provides the capabilities most needed in a typical debugging, development session. Currently, we have been using X Release 10 for such programs. In the future, we expect to be using Release 11 which is expected to become the industry standard for window systems.

4 Controlling the Utah-MIT hand

In this section, we provide a short overview of how the *Condor* is used to control the Utah-MIT hand, an advanced robot that has been developed by the Center for Engineering Design at the University of Utah and the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology to study issues involved in machine dexterity. There have been a series of papers published on different aspects of this project; on design and construction of the hand (Jacobsen et al. [1984] and [1986]), on low-level control issues (Biggers et al. [1986]), on algorithms for force control (Hollerbach et al. [1986], and on tactile sensor design and construction (Siegel et al. [1987]).

4.1 The Controller Hierarchy

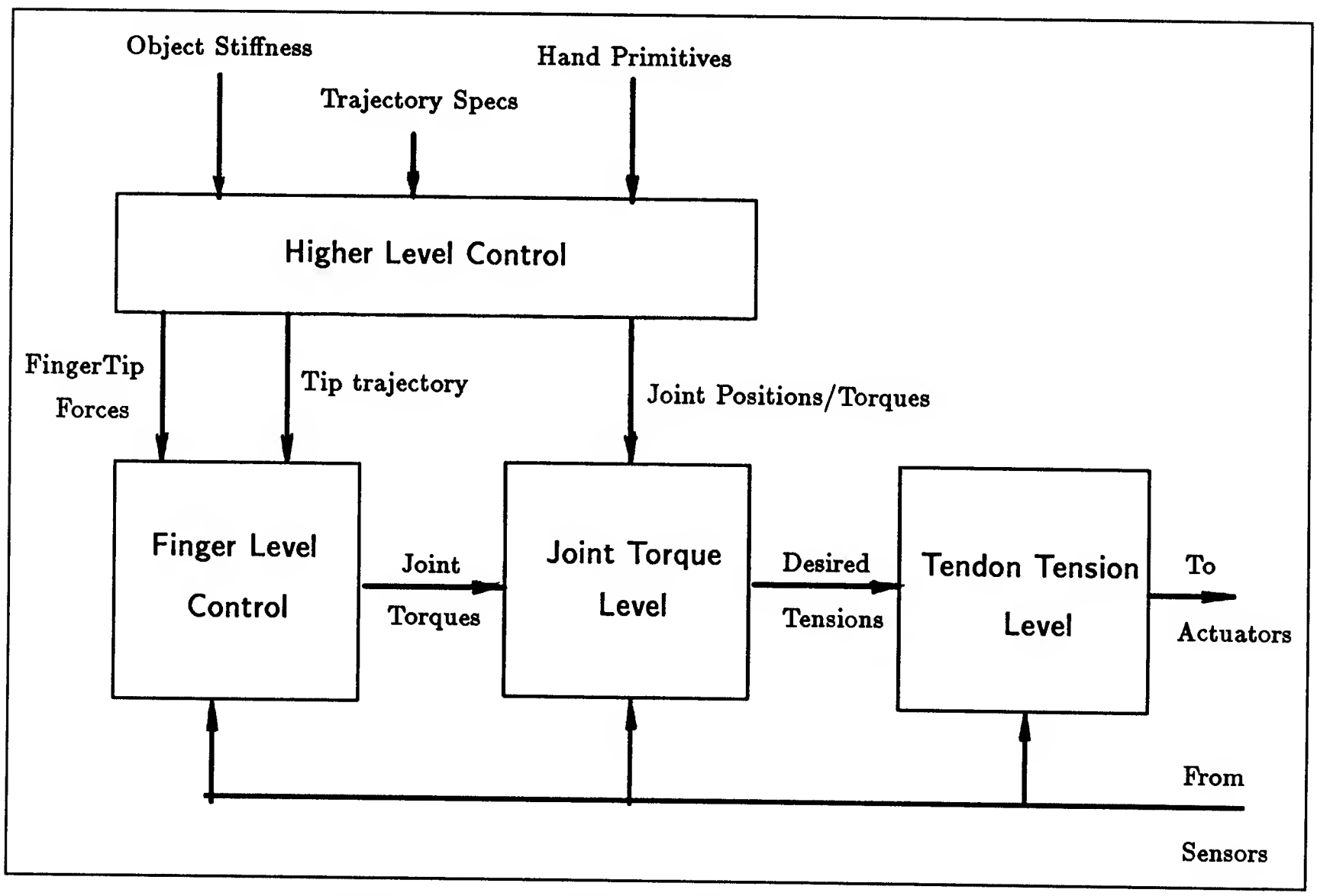

Figure 3: Block diagram of the controller

The block diagram for the current controller hierarchy is indicated in Figure 3. The lowest level in the hierarchy is formed by the tendon space controller that takes as input the torque that needs to be exerted at the joint level. This level converts the torque to two

desired tendon tensions, appropriately rectifying and filtering the output to the actuators, as needed. The next level is formed by the joint torque servos that take as input either joint positions or joint torques. This level computes the torques that need to be exerted by the hand's joints, to achieve particular positions or torques taking into account the coupling between the various joints.

A higher level of the controller currently can operate in one of two modes; In the first mode, it takes as input *hand primitives* which are motion specifications of a number of the hand's joints. This level of the controller then converts these primitives to a stream of joint angle trajectories, to be achieved by the lower level controller. The user can directly specify these trajectories too. In the second mode, this level of the hierarchy takes as input a desired *grasp stiffness matrix.* Using this matrix, it can compute the torques to be exerted at the joints to correct for deviations from an expected grasp position of a grasped object.

4.2 Controller Implementation

The current controller implementation uses four processors. One processor is allocated for performing the joint-torque and tendon tension level servoing of eight joints at 400 hertz. Another processor is dedicated to controlling the *xyz* table. The fourth processor is a master controller.

An abstraction similar to that used for device configuration is also used for configuring the hand control programs. There is one central configuration file, which specifies which joints are allocated to which processor. This configuration file also specifies which A/D's and D/A's are allocated to a particular joint. This permits hand control programs to be developed in a modular fashion. Changes in hardware occur in terms of changed connectors, boards, and hardware addresses. To cope with these changes, no changes to the actual control programs is necessary. All that is needed are small changes to the controller configuration file and recompilation of the hand controller system.

The trajectory generator currently executes at 50 Hz on the master processor. Besides executing this slower loop, the master processor also performs a number of other functions.

(a) It forms the primary interface to the Sun-3, to load and save entire sequences of motions.

(b) It provides a sophisticated user interface to the user, where a user can select a subset of joints to monitor at any given moment. The required controller variables can be tuned from this level, and the status of the hand at any given moment can be ascertained.

(c) Controlling the execution of the servos on the slaves. Using the message passing system, it can enable or disable servos on the other processors, find out their status, or restart errant programs.

Besides the actual real-time programs, the hand control environment also comprises of programs that run on the Sun-3. These programs include a real-time plotter that allows a user to monitor any of the controller variables in real time, and a kinematic simulator that

can be used to generate input to the trajectory generator running on the master processor. We have also concluded implementing an RPC based interface to such a controller that enables trajectories to be enqueued from a Lisp Machine.

5 Conclusion

This paper has described a controller architecture suitable for the demanding computational tasks of robots like the Utah-MIT hand. Efficiency and convenience are the key features that distinguish this controller from other real-time systems; the *Condor* lacks the bells and whistles that lead to inefficiency that are present in many real-time architectures, but it provides convenient and powerful mechanisms for code development. Since the hardware is built out of standard components, such a system can be built relatively easily. We hope that this will allow other robotics researchers to concentrate on the tasks they wish to be spending time on, by reducing the time they currently spend on designing, building and debugging custom hardware for robotics related projects.

To summarize, the *Condor* software includes the following key features:

(a) No dynamic loading of device drivers.

(b) An efficient servo-loop scheduler for tasks that need such a facility.

(c) Efficient low level communication primitives built upon a shared memory hardware base.

(d) Easy methods for circumventing message passing when even higher speed communication is required.

(e) Implementation of `stdio` for both terminal devices and disk files.

(f) Efficient mathematical and control library routines.

(g) Built in support for debugging.

The hardware redesign described in this paper reflects our technical experiences with the earlier *Muse* architecture; the software has been made more streamlined, portable and efficient, and the environment has been enhanced. During the past year, we have been actively using the *Condor* for program development. We are now reasonably satisfied that the *Condor* provides a researcher with the flexibility needed to write programs that are typical of advanced robotics research, without sacrificing performance.

6 Acknowledgements

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the AI-Lab's artificial intelligence research is provided in part by the Office of Naval Research University Research Initiatives Contract N00014-86-K-0180 and the Defense Advanced Research Projects Agency under Office of Naval Research contracts N00014-85-K-0124.

The authors would also like to acknowledge the contributions made by the following people to the building of the *Condor* system. The first version of our architecture was based on work done for the Concert Project under Prof. Halstead at MIT's Real Time Systems Group. David Kriegman wrote the first version of our message passing system. David Taylor wrote the evh (exception vector handler) device to trap unhandled vectors on the Sun-3, and wrote the `ptrace/wait` emulator, which helped in porting the debuggers to our environment. Steve Drucker wrote some of the early X 10 code, and has always been around to help us deal with all the cabling.

7 References

1. **Biggers, K. B., Gerpheide, G. E., Jacobsen, S. C.,** "Low-level control of the Utah-MIT dextrous hand," *IEEE Conference on Robotics and Automation*, pp. 61-66, San Francisco, April 1986.

2. **Bisiani, R., and Reddy, R.** "Workshop on Computer Engineering for Robotics", *The Robotics Institute*, Carnegie-Mellon University, Pennsylvania, October 1982.

3. **Brooks, T., and Wilcox, B.,** "Workshop on the Application of VLSI for Robotic Sensing", *Jet Propulsion Laboratory*, Vol. 1, March 1984.

4. **Chen, J. B., Fearing, R. S., Armstrong, B. S. and Burdick, J. W.,** "NYMPH: A Multiprocessor for Manipulation Applications," *Proc. IEEE International Conference on Robotics and Automation,* pp. 1731-1736, San Francisco, April 1986.

5. **Gauthier, D., Freedman, P., Carayannis, G., and Malowany, A. S.,** "Interprocess Communication for Distributed Robotics," *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 6, pp. 493-504, 1987.

6. **Hollerbach, John M.,** "Kinematics and dynamics for control," in: *SDF Benchmark Volume on Robotics*, M. Brady, ed., MIT Press, Cambridge, Massachusetts, 1987, in press.

7. **Hollerbach, J. M., Narasimhan, S., Wood, J. E.,** "Finger force computation without the Grip Jacobian," *IEEE Conference on Robotics and Automation*, pp. 871-875, San Francisco, April 1986.

8. **Jacobsen, S. C., Wood, J. E., Knutti, D. F., Biggers, K. B.,** "The Utah/MIT Dextrous Hand: Work in Progress," *Robotics Research*, MIT Press, pp. 601-653, Cambridge, MA, 1984.

9. **Jacobsen, S.C., Iversen, E.K., Knutti, D.F., Johnson, R.T., and Biggers, K.B.,** "Design of the Utah/MIT Dextrous Hand," *Proc. IEEE Conference on Robotics and Automation,* pp. 1520-1532, San Francisco, April 1986.

10. **Kazanzides, P., Wasti, H., and Wolovich, W. R.,** "A Multiprocessor System for Real-Time Robotic Control: Design and Applications", *IEEE Conference on Robotics and Automation*, pp. 1903-1908, Raleigh, NC, April 1987.

11. **Kim, J. J., Blythe, D. R., Penny, D. A., Goldenberg, A. A.,** "Computer Architecture and Low Level Control of the Puma/RAL Hand System: Work in Progress," *IEEE Conference on Robotics and Automation*, pp. 1590-1594, Raleigh, NC, April 1987.

12. **Leung, S. S., and Shanblatt, M. A.,** "Real-Time DKS on a Single Chip," *IEEE Conference on Robotics and Automation*, pp. 453-456, Raleigh, NC, April 1987.

13. **Leung, S. S., and Shanblatt, M. A.,** "Computer Architecture Design for Robotics", *IEEE Conference on Robotics and Automation*, pp. 453-456, Philadelphia, April 1988.

14. **Lozano-Perez, T., Brooks, R.** "An approach to Automatic Robot Programming," *Artificial Intelligence Laboratory Memo AIM 842*, MIT Artificial Intelligence Laboratory, April 1985.

15. **Narasimhan, S., Siegel, D. M., Hollerbach, J. M., Biggers, K. B., Gerpheide, G.E.,** "Implementation of Control Methodologies on the Computational Architecture for the Utah/MIT Hand," *IEEE Conference on Robotics and Automation*, pp. 1884-1889, San Francisco, April 1986.

16. **Narasimhan, S., Siegel, D. M.,** "The Condor Programmer's Manual - Version II", *A. I. Lab. Working Paper No: 297*, July 1987.

17. **Paul, R. P., and Zhang, H.,** "Design of a Robot Force/Motion Server", *IEEE Conference on Robotics and Automation*, pp. 1878-1883, San Francisco, April 1986.

18. **Salisbury, K.,** "Kinematic and force analysis of Articulated Hands," *Ph. D. Thesis*, Department of Mechanical Engineering, Stanford University, July 1982.

19. **Siegel, D.M., Narasimhan, S., Hollerbach, J.M., Kriegman, D., Gerpheide, G.,** "Computational Architecture for the Utah-MIT Hand," *IEEE Conference on Robotics and Automation*, pp. 918-925, St. Louis, March 1985.

20. **Siegel, D.M., Garabieta, I., Hollerbach, J.M.,** "A capacitive based tactile sensor," *Proc. of the SPIE Conference on Intelligence Robots and Computer Vision*, pp. 153-161, Cambridge, MA, September, 1985.